Electronic Notes in Theoretical Computer Science 82 No. 5 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 13 pages

A Low-Level Analysis Library for Architecture Recovery

Welf Löwe [1]

Software Tech. Group, MSI
Växjö University
351-95 Växjö, Sweden

Jonas Lundberg [2]

Software Tech. Group, MSI
Växjö University
351-95 Växjö, Sweden

Abstract

We discuss low-level program analyses that are reusable in many high-level analyses for architecture recovery. They include both program structure analyses known from compiler front-ends and data-flow analyses. In general, high-level analyses use the low-level results only partially. We argue that it is necessary to perform the low-level analyses completely, though. To deal with large systems and to adapt to different high-level analyses, it is, however, necessary to control the low-level analyses' precision and scope.

1 Introduction

The detection and extraction of components from a legacy system is one way of reusing successful components that have not been designed for reuse. Unfortunately, the architecture, i.e. the components and connectors, is hardly documented in such systems. The only trustworthy source of information is the system implementation. Hence, the components have to be retrieved from this source. As real world legacy systems tend to be large, the source code cannot be read directly. Instead, one uses semi-automatic program analyses to extract the information.

[1] Email: welf.lowe@msi.vxu.se
[2] Email: jonas.lundberg@msi.vxu.se

It is common sense that architecture recovery of legacy systems benefits from precise data-flow analyses. For example, a precise type or points-to analysis would lead to more precise call or class graphs, the basis of many of these analyses. However, the data-flow analyses are difficult to implement, often more difficult than the analyses for architecture recovery themselves. One might argue that good could be good enough: too exhaustive data-flow analyses are too expensive and, as architecture recovery is a heuristics anyway, they would not be justified by the results. Even more so as legacy systems are large and the complexity of the (context-sensitive) data-flow analyses grows exponentially with their precision. However, we lack comparative studies relating the quality of the architecture recovery results to the precision of the data-flow analysis.

Similar observations continue to hold for many other re-engineering and forward-engineering techniques like detection of individual components, software comprehension, reverse engineering, and software validation or optimization. Data-flow analyses are considered helpful but do not seem to justify the effort.

We propose a low-level analysis library reusable in different high-level analysis contexts. It is tunable in precision in order to easily perform experiments answering the question: How precise is precise enough for a certain analysis problem.

We organize the paper in the following way: Section 2 describes problems in architecture recovery and analyzes why data-flow analyses are essential. It also sketches the currently proposed analyses for architecture recovery. Section 3 gives an overview of those compiler front-end and data-flow analyses that we refer to as low-level analyses and that should be provided in the library. It also discusses the practicability of the current analyses. Section 4 addresses the problem of complexity and precision of low level analyses. Section 5 presents our approach.

2 The Architecture Recovery

The goal of software architecture recovery is to recover an abstract architecture representation of a given system. A system's architecture is defined by its components, communications among them and the containment relation of components in larger components [3] [11,25,5,21,24,19]. We define components to be software artifacts with typed input and output ports. This definition focuses on computational components, but is sufficiently general to cover all other variants. Components are identified as larger units of "coherent" modules or classes. The notion of coherency usually mixes static and dynamic system properties: it includes structural connection among the modules or classes in the call or inheritance graphs (static information). Additionally,

[3] The notion of components is recursive.

it requires strong interactions between the modules or classes by actual calls (behavioral information).

In a software architecture, the essential communication among components is given in terms of ports and connectors. A port defines points in a component that provide data to its environment and require data from its environment, respectively. A connector defines out- and in-ports to be connected and specifies whether data is transported synchronously or asynchronously [14]. Some connectors may be as complex as most components, and thus require the same amount of consideration in recovery.

In practice, ports and connectors are implemented by patterns using basic communication constructs like calls, RPCs, RMIs, input/output routines etc. provided by the implementation language or the component system. The Observer Pattern, e.g., is such a port and connector implementation as it connects an event generator with some listener objects. Hence, we must be able to identify these communication patterns in order to recover the architecture view of a system's inter-component communication (i.e. the connectors). Again, static structure analysis alone is insufficient. It often comes up with misleading results regarding source and target of the communication: Assume, e. g., communication is implemented by an event-listener pattern. The source of the communication provides a method called by the target to add itself as an event listener. Moreover, the event source captures the listeners in a container of abstract listener objects. There is usually no static type information pointing back to the communication target. This connection is only visible via the object identifiers captured in the communication source – such information is behavioral information.

Architecture recovery means in practice to identify individual components and communication patterns. As sketched above, both components and essential communication among them, are defined in terms of (static) structure and (dynamic) behavior. Hence, architecture recovery require both static and behavioral information in order to be successful.

Analyses for architecture recovery usually use a call or class graph of the system. A *call graph* is a directed graph $G_{call} = (V, E)$ with a vertex $v_m \in V$ for each method or attribute m in the system and a directed edge $(v_m, v_n) \in E$ for each m potentially accessing method or attribute n. Polymorphism is resolved by explicit, monomorphic dispatch method residing in the abstract classes. Attribute access is resolved by set and get methods. A *class graph* is a directed graph $G_{class} = (V, E)$ with a vertex $v_c \in V$ for each class c in the system and a directed edge $(v_c, v_k) \in E$ if G_{call} of that system contains an edge (v_m, v_n) with m a method of c and n a method or attribute of k.

The computation of call and class graphs require some low-level analysis to derive abstractions involving classes (or objects) interconnected by various dependencies (e.g. inheritance, method calls, and aggregation).

Then there are different techniques to identify individual components, connectors, or well-known design patterns – actually possible candidates thereof.

Established techniques to identify component candidates are different clustering techniques based on metrics, dominance analysis or concept analysis, as well as pattern detection used to identify well-known, and often used, communication patterns. The state of the art is summarized in [18].

The basic approaches in the above techniques are similar in spite of their separate appearances. It is a bottom-up approach starting from the call or class graph. Then they "collapses" coherent entities into a single vertex, using the above mentioned techniques to identify components, or collapse vertices and edges to a single edge using techniques to identify connectors (i.e. communication patterns). As notion of components is recursive; this process continues until the whole graph is collapsed.

3 Low Level Analyses

We denote the classic analysis techniques known from the field of compiler construction as low level. Their origin is the translation and optimization of programs. However, they can support the understanding of systems on higher levels of abstraction than the source code. We distinguish analyses for checking the well-definedness of programs from behavioral analyses of program runs. The former is done by compile front-ends the latter by data flow analyses. They are used to construct call and/or class graphs of the system.

Scanners, parsers and static semantic analyzers perform basic program analysis. Meta-programming libraries provide APIs to access their results, which are basically attributed syntax trees (ASTs).

In the context of object-oriented languages with dynamic dispatch, the crucial step in constructing a call graph is to compute a precise approximation of the set of methods that can be invoked by a given method call. Hence, ASTs (and its definition table) can only provide a rough approximation of call and class graphs.

Many different call graph construction algorithms that have been proposed use data-flow analyses. They range from simple and fast (with relative low precision) [4,27] to more elaborated ones requiring data flow analysis [1].

On this level, classes are sometimes cloned to distinguish different subsets of instances (objects) of classes and their interaction. The latter type of class level representation is today rarely used since it requires extensive (and very memory consuming) data-flow analysis in order to derive a low level representation that can separate different instances of a class. Exceptions are the object graph presented in [26] and the object model presented in [22].

Data-flow analyses rely on the theory of monotonous data flow frameworks [20]. In each node of the basic block graph, we compute a analysis value of a lattice $L = (M, \sqsubseteq)$ of possible values M. Each block i is assigned a monotonous transfer function $f_i : L \to L$ representing an abstraction of the code in that basic block wrt. to the data-flow analysis problem. Initially, an appropriate value from L is assigned to the the start (in backward problems

end) block and $\perp$ to the other nodes. The analysis is then performed iteratively by actualizing the values $v_i \in L$ of nodes i where $v_i = f_i(sup_{j \in pred(i)} v_j)$, with $pred(i)$ as the set of predecessor nodes of i in the graph.

Although the classic basic block graph is defined procedure by procedure, data-flow analyses can also be performed in an inter-procedural way. Therefore, the basic block graph is extended to an inter-procedural control flow graph containing additional edges between call node and callee basic block graph and back. A problem with this extension is that some paths are never executed at runtime, e.g. paths from a call x to the callee and back to another caller y, but contribute to the results of the analysis.

Context sensitive analyses [13] avoid the negative influences of non-executable paths on the analysis results. They distinguish analysis results reaching a node via different paths, approximated by the path's last k nodes. Contexts may be derived from previous analysis information: [15] and [12] define an analysis context as the result of a context insensitive analysis.

Context sensitive and inter-procedural data-flow analyses can also be performed on Static Single Assignment (SSA) form representations of the program, reducing the analysis time and representation size compared to basic block graphs. SSA representations store provable equivalent computation nodes only once. All uses of their values refer to this single representation. If more then one definition of a value reaches (potentially) a use, the join of the definition is represented by a pseudo computation, a so called ϕ-function. With this trick, each use refers to exactly one definition.

[10] defines the standard algorithm for computing the SSA form. By combining this with value numbering [9], and dead code elimination [7,6], the number of ϕ-function can be reduced. [2] showed that constant propagation, arithmetic simplifications, dead code elimination and elimination of partial redundancies together with SSA construction not only leads to more compact representations but also to shorter compilation times. The additional analyses are amortized as they enable fewer memory allocation operations.

The generalization to Heap-SSA by Martin Trapp [28] allows to represent object-oriented programs: computations with side effects on the memory consume and compute values of memory type allowing to model dependencies and anti-dependencies of these operations.

However, the current state of the art in data-flow analysis is not satisfactory regarding architecture recovery. There are open problems in both program analysis and representation of analysis results. The memory consumption of analysis results is still the main problem for handling large systems. However, we observe redundancies, especially in the representation of results of context-sensitive analyses as often distinguished contexts do not provide different analysis results. The exceptions [3,28] only control small example programs. The analyses themselves are either too inefficient or too ineffective (sometimes even both). The reason is a statically predefined precision of the analyses with the exceptions of [23,28]. Interleaving different analy-

ses improves the overall precision. This is disregarded by many approaches, attempts [8] and more elaborated [28,16].

4 Analysis Precision and Complexity

Current architecture recovery analyses usually deal with structural information only as they are targeted to large systems. Because of their complexity, precise data-flow analyses are simply not applicable [13].

To compensate for this lack of dynamic information, some approaches use profile information from system runs. However, this only provides information that is strongly depending on the profiled use cases and, even worse, on the used data sets in the use cases. It is therefore unclear, how the profile information generalize.

In some cases, however, data-flow analyses are too precise and detailed for the actual problems in software comprehension as observed by [26]. They provide a lot more information than actually needed on higher level.

In general, the final result of the high-level analysis could very much depend on the accuracy of the underlying low level analysis. E.g. the precision of the underlying call graph will affect the resulting class graph. By precision, we here mean the accuracy of the low level analysis used to construct the call graph, which we fold to get a class graph. The relative precision can easily be measured once a call graph have been derived by counting the number of edges and/or the number of reachable methods in the graph. In both cases, a low number indicates high precision since we always approximates the graph conservatively. [4] The precision of the call graph can also be estimated without any measurements. This is an approach taken by by Grove *et al* in [13], and to a lesser extent, by Tip and Palsberg in [27]. They use a lattice-theoretic model where each element of the lattice corresponds to a possible call graph for a program, and call graphs in the lattice are ordered in terms of relative precision. The interesting part of this work is that they have identified a number of parameter that influences the precision of a call graph construction algorithm. By comparing these parameters for different algorithms they can induce a partial ordering of the call graph algorithms.

The precise definition of the folding process makes it possible to transfer the results regarding the precision of the call graphs to class graphs. A call graph G^1_{call} with higher precision than another call graph G^2_{call} will always be folded to a class graph G^1_{class} with higher or the same precision as class graph G^2_{class} corresponding to G^2_{call}. This idea is depicted in Figure 1 where G^i_{comp} represents the component graph as the analysis results based on class graph G^i_{class} folded from call graph G^i_{call}.

[4] Conservative approximations in the call graph algorithm are necessary when the call graph is going to be used for compiler optimization. In program analysis aimed at program understanding, this is not necessarily the case. A "good" optimistic call graph may do a better work than a "bad" pessimistic.

$$\begin{array}{ccccccc}
G^{i}_{call} & \subset & G^{j}_{call} & \subset & G^{k}_{call} & \cdots \\
\Downarrow & & \Downarrow & & \Downarrow & \\
G^{i}_{class} & \subseteq & G^{j}_{class} & \subseteq & G^{k}_{class} & \cdots \\
\Downarrow & & \Downarrow & & \Downarrow & \\
G^{i}_{comp} & \subseteq & G^{j}_{comp} & \subseteq & G^{k}_{comp} & \cdots
\end{array}$$

Fig. 1. Result of the high-level analysis depends on the precision of the underlying program representations.

The question that remains to answers is to what extent the difference in precision will influence the result of the architecture recovery process. As indicated in Figure 1, a high precision of the underlying analysis will result in better high level analysis. However, it is likely that the influence of a high precision call graph gets diminished for each abstraction along the way to the final analysis.

We have two problems to identify the right precision of the low-level analyses:

(i) What is feasible? This depends on the actual system (size and structure) to analyze and on the effort that one can tolerate in the individual recovery process.

(ii) What is necessary? This could even differ for different parts of the same system.

Hence, the precision must be adaptable for the user of a low-level analysis library just by parameterization.

5 General Approach

We use a meta programming front-end for the AST construction and the approach of [28] to construct an SSA representation of the system with the optimizations of [2] to reduce its size. This is available as a library "libFirm" already [17].

The low-level analysis library performs value numbering data-flow. For pointer typed expressions, this is pointer analysis; for type tags, type analysis. In general, each type of values T corresponds to an initial value lattice $L_T = (M_T, \sqsubseteq_T)$; each SSA node type n gets $outdegree(n)$ initial transfer functions:

$$f^n_i : L_{T^{in}_1} \times L_{T^{in}_2} \times \ldots \times L_{T^{in}_{indegree(n)}} \rightarrow L_{T^{out}_i}$$

with $T^{in}_1, T^{in}_2 \ldots T^{in}_{indegree(n)}$ types of input edges and T^{out}_i the type of the i-th output edge of n.

All low-level analyses depend on another. One cannot perform precise constant analysis without type analysis without pointer analysis without control flow analysis without constant analysis. (The ordering in that enumeration

does not matter). Hence, even if we are only interested in the precise types of some expressions or in the targets of some pointers to construct a call graph, we cannot get around to solve the other analysis problems. Otherwise, we lack precision in general; it could not be increased just by parameterization. Instead, one has to think up the kind of values ignored so far, define the corresponding data-flow problem, attach corresponding transfer functions to the nodes of the intermediate representation, implement a corresponding value lattice, and redefine any other analysis implemented so far to take advantage of the freshly introduced values.

In order to apply the low-level analyses to large systems, they are adaptable in precision and the slice of the program they analyze.

5.1 Adapt the Analysis Precision

There are two ways to increase precision:

(i) Refine the $L_T = (M_T, \sqsubseteq_T)$ and the corresponding transfer functions,

(ii) Introduce, and stepwise, increase the context sensitivity of the analyses.

For both we follow the ideas of Martin Trapp [28] used for optimizing object-oriented programs.

The refinement of the lattices and the corresponding transfer functions are critical as "by-hand" redefinitions are to avoid. Instead, we generalize the lattices $L_T = (M_T, \sqsubseteq_T)$ to $L_T = (\mathcal{T}(M_T), \sqsubseteq_T)$ with $\mathcal{T}(M_T)$ a term algebra over M_T. Using algebraic identities, we could increase precision of analyses.

Example 5.1 Let $T = \texttt{int}$ and $L_{\texttt{int}}$ the constant lattice of over $\{\texttt{minint}, \ldots, \texttt{maxint}\}$. Furthermore, assume SSA nodes $plus, minus$ with indegree 2 and outdegree 1 and the usual semantics. Their transfer function are defined as follows:

$$f_1^{plus}(i,j) = i + j \qquad \textit{if } i,j \textit{ integer constants}$$
$$f_1^{plus}(\top, \_) = f_1^{plus}(\_, \top) = \top$$
$$f_1^{minus}(i,j) = i - j \qquad \textit{if } i,j \textit{ integer constants}$$
$$f_1^{minus}(\top, \_) = f_1^{minus}(\_, \top) = \top$$

Furthermore, assume an SSA fragment as depicted below:

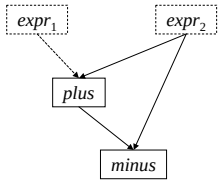

Fig. 2. SSA fragment from $expr_1 + expr_2 - expr_2$ for some integer expressions $expr_1, expr_2$.

If the value analyzed for either of $expr_1, expr_2$ is $\top$, i.e. unknown, $expr_1 + expr_2 - expr_2$ gives $\top$ as well, even if $expr_1$ is a constant.

Let $\mathcal{T}$ be a term algebra over `int` with $\{\texttt{minint}, \ldots, \texttt{maxint}\} \subset \mathcal{T}$ and $t_1, t_2 \in \mathcal{T} \Rightarrow \texttt{plus}(t_1, t_2), \texttt{minus}(t_1, t_2) \in \mathcal{T}$ and the usual algebraic identities. Let $L_{\texttt{int}}$ be the constant lattice of over the (constant) ground terms in $\mathcal{T}$. The transfer functions are now:

$$\begin{aligned} f_1^{plus}(i,j) &= i + j && \textit{if } i,j \textit{ integer constants} \\ f_1^{plus}(I,J) &= \texttt{plus}(I,J) \\ f_1^{minus}(i,j) &= i - j && \textit{if } i,j \textit{ integer constants} \\ f_1^{minus}(I,J) &= \texttt{minus}(I,J) \end{aligned}$$

Let $Expr_1, Expr_2$ be the terms analyzed for $expr_1, expr_2$. The result of the same fragment is now $\texttt{minus}(\texttt{plus}(Expr_1, Expr_2), Expr_2)$ i.e, using algebraic identities in $\mathcal{T}$, $Expr_1$.

In general, the term algebras $\mathcal{T}_T$ for each value type T are constructed systematically. Each element of M_T is a term in $\mathcal{T}_T$. Moreover, we introduce operator symbols `n` for each SSA node n with an output of type T to construct terms of $\mathcal{T}_T$. The new transfer functions just construct new terms from the argument (terms).

Terms increase the precision of the analysis. However, as the terms are unbound in depth, the analyses would not terminate. Eventually, we have to cut them. This is performed using the originally transfer functions.

Example 5.2 A term $\ldots\texttt{plus}(\texttt{plus}(\texttt{plus}(Expr_1, Expr_2), 1), 1\ldots$ could easily be reduced to $\top$ using the original definition of the transfer function f^{plus}.

The result of the above construction is that the precision is just a *parameter*, namely the allowed depth of the terms before the original transfer functions are applied leading to a widening of information and a reduction of term size. This parameter can be increased automatically.

Initially the library performs simple context-insensitive analyses. On demand, they can be refined to advanced context-sensitive analyses with a stepwise refinement of distinguished contexts value types. Context-sensitive analyses distinguish analysis results reaching a node via different paths, approximated by the path' last k nodes. This is already parameterized precision.

As path and values are unified only at ϕ nodes, we only keep track of values reaching a node via different entries of directs or transitively depending ϕ nodes. This does not need any special consideration. We only have to keep the effects of the join functions symbolically as long as the depth of the terms does not exceed our precision parameter.

The non-symbolic and symbolic definition of the ϕ nodes are:

(1) $f^{\phi}(I_1 \ldots I_n) = sup(I_1 \ldots I_n)$

(2) $f^{\phi}(I_1 \ldots I_n) = \chi(I_1 \ldots I_n)$

with χ a new symbol of the corresponding term algebra.

Example 5.3 Assume an SSA fragment as depicted below:

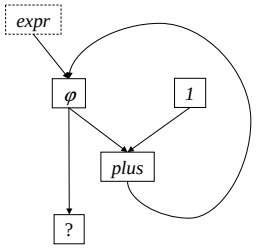

Fig. 3. SSA fragment from $while(i = expr; cond; i++)$ for some unspecified expression *expr* and condition *cond*.

The analysis value before "?" is $\top$ for the original constant lattice using transfer function (1) and would infinitely construct terms:

$$\chi(Expr, \texttt{plus}(\chi(Expr, \texttt{plus}(\chi(Expr, \texttt{plus}(\chi(\ldots), 1)), 1)), 1)), 1))$$

using transfer function (2). Depending on the depth of terms it would be

$$\begin{aligned}
\top &: insensitive \\
\chi(Expr, \texttt{plus}(\top, 1)) &: 1\ symbolic\ \phi \\
\chi(Expr, \texttt{plus}(\chi(Expr, \texttt{plus}(\top, 1)), 1)) &: 2\ symbolic\ \phi \\
\chi(Expr, \texttt{plus}(\chi(Expr, \texttt{plus}(\chi(Expr, \texttt{plus}(\top, 1)), 1)), 1)) &: 3\ symbolic\ \phi \\
&\ldots
\end{aligned}$$

Simple algebraic identities would simplify the terms to

$$\begin{aligned}
\top &: insensitive \\
\chi(Expr, \top) &: 1\ symbolic\ \phi \\
\chi(Expr, \chi(Expr + 1, \top)) &: 2\ symbolic\ \phi \\
\chi(Expr, \chi(Expr + 1, \chi(Expr + 2, \top))) &: 3\ symbolic\ \phi \\
&\ldots
\end{aligned}$$

Obviously, the depth of the terms reflects the context sensitivity of the analysis.

In order to guarantee that same terms compute the same values, we must index the χ symbol with the corresponding ϕ node id.

5.2 Adapt the Analysis Focus

Orthogonal to the stepwise refinement of the analysis, the library allows to focus on subsystems. There are two ways to change focus:

(i) Switch from intra-procedural to inter-procedural analyses.

(ii) Analyze partitions of the systems separately with different precision.

In the intra-procedural analysis, all values entering a procedure are assumed to be $\bot$. The analysis takes no advantage of results of the analysis of its environment. As these results are not to propagate, the intra-procedural analysis is very efficient. Switching to inter-procedural just removes the restriction of not propagating results.

Instead of inspecting the whole system with the same precision, it should be possible to analyze partitions thereof separately with different precision. Since different precisions just refer to a different depth of terms, propagation of analysis values across the borders of different precision is simple: values (terms) propagated from a lower to a higher level of precision continue to be valid. Values propagated from a higher a to lower level of precision are simplified as usual when they exceed the accepted depth.

After each new analysis, succeeding optimizations, e.g. dead code elimination, reduce the code size such that that more advanced analyses become possible. Moreover, we perform transformations reducing artificial dependencies in the SSA graph, e.g. removing edges representing not existing memory dependencies and calls.

6 Conclusion

We proposed and discussed a low-level analysis library, parameterizable in precision and adaptable in focus. It is used to support high-level analyses for architecture recovery, mainly with data-flow analyses. Therefore, we thoroughly explored the state of the art in low-, and high-level analyses. We derived requirements from the intended high-level analyses and assembled suitable low-level analyses techniques.

Currently, we cannot answer the questions how precise the low-level analyses ought to be – to improve the high-level results – and may be – to be still feasible. Therefore, our next steps are implementing the library and performing experiments. On the long run, we hope that this low-level analysis library generalizes for other high-level analyses and optimizations.

References

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*, pages 2–26, 1995.

[2] Markus Armbruster and Christian von Roques. Fiasco – fiasco is a sather compiler. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, 1996.

[3] Uwe Assmann and Markus Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In S. Giloi, W. K., Jähnichen and B. D. Shriver, editors, *Proceedings of the Conference on Programming Models for Massively Parallel Computers*, pages 74–82, Los Alamitos, CA, USA, September 1993. IEEE Computer Society Press.

[4] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Notices*, 31(10):324–341, 1996.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[6] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, 1998.

[7] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 55–66, 1991.

[8] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[9] Clifford Noel Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, Houston, Texas, February 1995.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 25–35, Austin, TX, USA, January 1989. ACM Press.

[11] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40, Singapore, 1993. World Scientific Publishing Company.

[12] W. Golubski. Typanalyse für objektorientierte programme, 1998. Habilitationsschrift an der Universität Siegen.

[13] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *OOPSLA*, 1997.

[14] Dirk Heuzeroth, Welf Löwe, Andreas Ludwig, and Uwe Aßmann. Aspect-oriented configuration and adaptation of component communication. In Jan Bosch, editor, *Third International Conference on Generative and Component-Based Software Engineering, GCSE*, page 58 ff. Springer, LNCS 2186, 2001.

[15] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural side effect analysis with pointer aliasing. *SIGPLAN Notices*, 28(6):56–67, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[16] Sorin Lerner, David Grove, and Craig Chambers. Combining dataflow analyses and transformations. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, volume 37(1) of *SIGPLAN Notices*, pages 270 – 282. ACM SIGACT-SIGPLAN, 2002.

[17] Götz Lindenmaier. libfirm – a library for compiler optimization research implementing firm. Technical Report 2002-5, Sep 2002.

[18] Welf Löwe, Morgan Ericsson, Jonas Lundberg, and Thomas Panas. Software comprehension - integrating program analysis and software visualization. In *Software Engineering Research and Practice (SERPS)*, 2002.

[19] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[20] Marlowe and Ryder. Properties of data flow frameworks. A unified model. *Acta Informatica*, 28:121–163, 1990. An overview of data flow frameworks and their characterizing properties is given. Contains many references to the field of data flow analysis.

[21] Ralph Melton. *The Aesop System: A Tutorial*. School of Computer Science Carnegie Mellon University, `http://www-2.cs.cmu.edu/~able/aesop/aesop_home.html`, 2002.

[22] Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Scholl of Computer Science, Carnegie Mellon University, Pittsburg, USA, 2000.

[23] John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings OOPSLA '94*, pages 324–340, October 1994.

[24] Shaw, DeLine, and Zelesnik. Abstraction and Implementations for Architectural Connections. In *3rd International Conference on Configurable Distributed Systems (ICCDS '96)*, 1996. concerns Unicon.

[25] M. Shaw and D. Garlan. *Software Architecture in Practice – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[26] Andre Spiegel. Object graph analysis. Technical report, 1999.

[27] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, 2000.

[28] Martin Trapp. *Optimierung Objektorientierter Programme*. Xpert.press. Springer, 2001. In German.